

Kennedy Grant
IN-61-CR
1828379
p. 22

Symbolic Inversion of Control Relationships in Model-Based Expert Systems

Final Report
GRANT NAG10-0045

Dr. Stan Thomas
Wake Forest University
Winston-Salem, NC

June 1988 - December 1988

(NASA-CR-182857) SYMBOLIC INVERSION OF
CONTROL RELATIONSHIPS IN MODEL-BASED EXPERT
SYSTEMS Final Report, Jun. - Dec. 1988
(Wake Forest Univ.) 22 p

N89-20631

CSCL 09B

Unclassified
G3/61 0188379

I. Overview

We have looked at symbolic inversion from several perspectives. First, we looked at a number of symbolic algebra and mathematical tool packages in order to evaluate their capabilities and methods, specifically with respect to symbolic inversion. Second, we ported the KATE system (without hardware interface) to a Zenith Z-248 microcomputer running Golden Common Lisp. The interesting thing about our port is that it allows the user to have measurements vary and components fail in a "non-deterministic" manner based upon random values from probability distributions. This type of environment has potential for using KATE for training purposes. Third, we thoroughly studied INVERT as currently implemented in KATE, documented its operation, identified some of its weaknesses and made some corrections to it. The corrections and enhancements are primarily in the way that logical conditions involving AND's and OR's and inequalities are processed. In addition, the capability to handle equalities was also added. It was found that INVERT returned incorrect results on several classes of expressions. We also made suggestions regarding the handling of ranges in INVERT. Last, we have looked at other approaches to the inversion process and made recommendations as to how future versions of KATE should perform symbolic inversion.

II. Computer Algebra and Symbolic Mathematical Software

It has been stated [Corn87] that "INVERT performs some of the same mathematical manipulations that Artificial Intelligence programs such as Mathematica and TK Solver do, but is less comprehensive." In order to explore further the nature of this similarity and to gain a better understanding of the working of symbolic algebra and "mathematical tool" programs, several such packages were acquired and compared. We will describe each of the packages examined individually and then their relationship to INVERT as a group.

Computer Algebra

Computer algebra systems perform symbolic mathematics, for example, solving $a \cdot x + b = c$ for x to yield $x = (c-b)/a$. In addition, most are capable of doing numeric computations such as evaluating the equation above at specific values of a , b , and c . Their capabilities often include algebraic simplification, polynomial factorization over the integers, solution of equations, integration of definite and indefinite integrals, and differentiation in several variables. Perhaps the best known computer algebra system is MACSYMA. Even though only widely available since 1983, its development began with Project MAC in the late 1960s. Today, more than sixty different computer algebra systems are known to exist [Hulz83]. We give a brief overview of three of the more widely used computer algebra systems: MACSYMA, Maple, and MuMATH.

MACSYMA

MACSYMA [MACS83, Rand84] was originally developed at MIT to run on the DEC-10 and was made available to other users via ARPANET. Since 1983, Symbolics Corporation has been responsible for its distribution and it is now available on several Motorola 68000 class machines, VAXEN, Symbolics and Honeywell. MACSYMA is a large (8.7 Mbytes of Disk, 3.25 Mbytes of RAM) system written in LISP. It is commonly accepted to be the most comprehensive and capable symbolic algebra package in existence today, even though some of the newer systems have more capabilities in terms of graphics and user interfaces. MACSYMA's capabilities for solving and simplifying equations are enormous but its forte is in the areas of integration, differentiation, and differential equations.

Maple

Maple [Char83, Char84] was developed at the University of Waterloo to be a compact, portable computer algebra system to be used on microcomputers in educational environments. It requires modest resources (3.1 Mbytes of Disk, 100 Kbytes of RAM) but is not really a stand-alone system since it does not even have built-in editing facilities. In spite of these constraints, it is capable of doing a wide variety of symbolic mathematics such as factoring, differentiation, integration, computing limits and series, matrix

manipulations and solving systems of equations. In all of these areas, Maple is more limited than MACSYMA. For example, Maple is not able to perform integration involving inverse trigonometric functions while MACSYMA has that capability. Maple does not currently solve ordinary differential equations.

MuMATH

MuMATH [Wilf82, Yun80] was the first symbolic algebra system developed for microcomputers. It was developed in a LISP dialect MuLISP. Like Maple, MuMATH requires only modest resources (600 Kbytes of Disk, 125 Kbytes of RAM), and like Maple, it has limited capabilities relative to MACSYMA. It has facilities for factoring, solving equations, integration and differentiation at the first-year calculus level. Perhaps the biggest selling point for MuMATH is that, like MACSYMA, the knowledgeable user can build very powerful tools by integrating the MuLISP language with the symbolic mathematical tools built into the system.

Mathematical Tools

In the 80's a new breed of software tool has become available which we will call "mathematical tools" for lack of a standard name for this type of software package. Marketing techniques often try to sell these tools as computer algebra systems when they are not. These tools are useful for solving systems of equations, plotting curves, evaluating functions, derivatives and definite integrals, but they do very little or no symbol manipulation. We will describe the capabilities of two such packages which we examined: MathCAD from MathSoft, and Eureka from Borland.

MathCAD

MathCAD [Math87] can be thought of as a scratchpad interface to a toolbox containing those numeric and graphing tools which are most useful to the practicing scientist or engineer. It can perform matrix operations, solve simultaneous equations and inequalities, compute derivatives and integrals, perform regressions, spline curve fitting, Fourier transforms and inverses and numerous other operations *as long as* the values involved are all numeric in nature. For example, MathCAD has no trouble evaluating

even the most complex double or triple integral but cannot do anything with even the most elementary indefinite integral. In general, MathCAD can only solve problems numerically, hence all answers must be numeric, they cannot be symbolic. This extends even to simplifying simple equalities such as $3 + x = 2*x - 4$. There is no way to express even this simple problem directly in MathCAD. The plotting facilities in MathCAD are excellent and easy to use as is the overall user interface. It has built-in routines for many functions including, among many others, Euler's gamma function $\Gamma(z)$, Kronecker's delta function $\delta(x_1, x_2)$, and Heaviside's step function $\Phi(x)$ and is a very useful tool for the applied mathematician or engineer already using a microcomputer in their work.

Eureka

Eureka [Eure87] is Borland's competitor for MathCAD. It costs only about one-third of the cost of MathCAD and probably has about one-half to two-thirds the functionality. The interface is very familiar to those who have used other Borland products but the system lacks the polish and professional touch of MathCAD. Given the overall similarity between the two products, Eureka has little to recommend it other than price.

Some Conclusions

The world of computer symbolic algebra is a rich, deep area for mathematical and computer science research [cf., Calm82]. We have not touched upon those difficulties here, only sampled some of the current products. All of the symbolic algebra packages mentioned have much more capability for performing computer algebra than does INVERT. Mathematically, the two are incomparable. The commercial products are generally distributed as compiled products and their techniques can only be, with the possible exception of MACSYMA, inferred or guessed. On the other hand, INVERT is capable of working with logical relationships expressed in LISP which are inappropriate for these other general-purpose systems. Is there anything to be learned from looking at these general-purpose systems? I think the answer is yes. I came away from this part of the project confident that INVERT is capable of handling the class of practical mathematical problems it needs to handle and that it can be enhanced with other capabilities as needed. Up to this point, most of the control functions handled by INVERT

are simple step functions or arithmetic expressions. Even non-monotonic and non-linear relationships could be handled by an improved notation and convention for representing ranges and multiple values in KATE. On the other hand, using these packages led me to believe that KATE could be improved by incorporating the use of UNITS into the computations carried out in INVERT. At the current time, the UNITS information stored in an objects frame is only used for display purposes. It would not be too difficult to have the operational routines in KATE such as INVERT process this information during computations. As a simple example, there is currently no check to ensure that the system is not adding feet and inches, or pounds and ounces. It may be argued that if the knowledge base is correct and standard units are used, such checks are not needed, but unit conversion checks could readily be implemented and would add additional integrity to KATE.

III. INVERT Software Description

At the highest level, INVERT can be viewed as a function which when given an equation and a variable solves the equation in terms of that variable. In the context of the operation of KATE, the inversion process begins by assembling a symbolic formula that expresses a sensor's expected value in terms of some other controlling object. This is done by composing an expression from the sensor's STATUS or SOURCE-PATH or SOURCE slot. The module INVERT then solves that expression for the controlling object given an observed or expected value [Scar87].

The primary variables used in INVERT are:

- RHS - The right hand side of the equation to be inverted.
- VAR - The variable to be solved for.
- LHS - The left hand side of the equation to be inverted. This is usually a constant and if it is numeric, it is automatically converted to floating point.
- FOUND - A sublist of RHS beginning with either the first appearance of VAR in RHS at the top level or the first sublist of RHS containing VAR at a lower level. FOUND is constructed by using a variant of SOME to map the utility function DEEP-FIND over RHS.

FOUND2 - FOUND2 is similar to FOUND except that it is a sublist of FOUND beginning with the **second** appearance of VAR in RHS. It is computed using only the CDR of FOUND, not all of RHS. A null value for FOUND2 indicates that VAR appears only once in RHS. *Well, that's the way it is supposed to work but it doesn't quite do that. For a RHS like (+ (- X X) 3) with X as VAR, FOUND2 will be nil even though X appears twice.*

The following comments apply to these variables and their use:

- 1) It is assumed that VAR appears once in RHS (except for CONDs where VAR may appear once in each COND clause but only in the first or last term of the clause). This assumption points out one of the current weaknesses in INVERT. Little or no standard simplification and rearrangement of expressions is done. Granted, simplification can be time-consuming but as KATE is used with larger and larger knowledge bases, this kind of feature found in general-purpose symbolic algebra systems will become necessary.
- 2) If no value of VAR can make the RHS equal the LHS, INVERT returns the special value ***INNOCENT***. *Actually, I find the use of the special value *INNOCENT to be very misleading and confusing as returned by INVERT. The value *INNOCENT* is sometimes returned to indicate that No value of VAR can satisfy the conditions of the inversion and in other situations it is returned as a signal that Any value of VAR satisfies the conditions. I find this very confusing.*
- 3) A subrange value is returned as a 3-element list of the form (***pair* low-value high-value**). This is another criticism of the current version of INVERT. It is too restrictive in its representation and handling of subranges. For example, there is no standard way to represent the fact that VAR can take on values in the range [0..3] OR [5..10]. In general, INVERT is sloppy about handling closed versus open intervals but that is probably because the measurements used are not themselves precise, often including tolerance values.
- 4) The value returned by INVERT may be an unevaluated expression such as (+ 4 2). It is not clear why terms are not evaluated when constructed.

Execution Details

The function INVERT proceeds as follows. It first checks for special and trivial cases such as null arguments or VAR not appearing within RHS. The recursion base case of equal RHS and VAR values is checked for at this point. INVERT then checks for

special RHS values such as COND's or an AND containing multiple occurrences of VAR. These are handled separately. The arithmetic operators +, -, *, and / are then processed by a recursive call to INVERT passing a new RHS which is the CAR of FOUND (the first term containing VAR) and a new LHS constructed from the "inverse" arithmetic operator from the first operator in the old RHS, the old LHS, and any other constants in the old RHS. This recursive call creates a new LHS which wraps the inversion of the operator currently under examination around the old LHS. Relational operators "<" and ">" are handled separately as are *if, and, or, not* and *max*. The remainder of this section focuses on the handling of these special cases.

COND's

To invert a COND, parameters consisting of a list of clauses, a VAR and LHS are needed. The operation of DO-COND is extremely convoluted and difficult to follow. I have studied it extensively and still do not understand what all of its special case considerations are intended to do. The most typical operation of DO-COND is to go through the list of COND clauses, looking for the first one containing VAR and whose returned value is equal to the value of the LHS value. When such a clause is found, its test condition is inverted recursively, seeking to find a value for VAR to make the value of the test equal to LHS. If no satisfactory clause is found, *INNOCENT* is returned.

AND Clause Containing VAR Twice

If an AND contains more than one appearance of VAR, then it is handled by overlapping the solutions of those terms in which VAR appears. As noted in a later section dealing with examples of errors found in INVERT, this is one of those areas in which INVERT can return incorrect answers because the value of the RHS is not considered in the solution returned.

Inequalities

To handle inequalities, three conditions are checked, the value of the LHS (called EV-LHS), whether or not the VAR being solved for appears immediately after the relational operator and whether the relation is less than or greater than. Less than and

greater than are the only two inequalities permitted. A simple boolean test of these three conditions is performed and a value returned. The testing of inequalities is another weakness of INVERT. The inequality is inverted solely in terms of VAR and no checking is done on the relationship between other constants in the expression. Also, the form of the expressions is unnecessarily restrictive, insisting that VAR appear as the first or last item in the expression. Examples of these problems are shown in a later section.

IF-conditions

In order to invert an IF, the test condition, the THEN-part, the ELSE-part, the first term containing VAR and the value of the LHS are used. The following tests are performed and a value returned. If the VAR term and test condition are equal, and the value of the LHS equals the THEN-part, then *t* is returned. If the VAR term and test condition are equal, and the value of the LHS equals the ELSE-part, then *nil* is returned. If the VAR term and the test are equal, **innocent** is returned. If the VAR term and the THEN-part are equal and the test evaluates to true, then **innocent** is returned. Otherwise, if the test evaluates to true then **innocent** is returned, otherwise the value of the else part is returned.

IV. Comments about INVERT

The following comments regarding INVERT are the most important part of this report as they illuminate several areas of weakness in the current operation of INVERT. Some, but not all, of these criticisms have been addressed and corrected in a modified version of INVERT submitted with this report. The comments are divided into two sections. The first section describes situations where INVERT appears to work correctly, that is, it does not appear to fail, but it returns misleading or incorrect results. The second section describes capabilities and kinds of problems that INVERT cannot currently handle at all but that we think are practical and should be done. In all of these comments we have tried to keep in mind the specialized environment in which INVERT operates and the fact that INVERT is not meant to be a general-purpose symbolic algebra inversion routine.

On the other hand, we expect that in the near future the knowledge base upon which INVERT relies may be machine generated or, at the very least, will be produced with less attention to detail than the prototype knowledge bases we have seen. It is necessary that INVERT be able to handle control relationships expressed in various forms and more flexibly than it currently operates.

Shortcomings in INVERT

The following examples are designed to illuminate areas in which INVERT returns erroneous, incomplete, or misleading results. Most of them have to do with inverting logical quantities and relations. INVERT does a very good job of inverting algebraic expressions containing the operators +, -, * and /. It is primarily in the realm of Boolean or logical quantities that INVERT needs a new, consistent and well-founded approach. The examples have been designed to illuminate general areas of weakness which we feel could easily be encountered with the type of control relationships used in KATE's knowledge base. They have been kept as simple as possible in order to focus the reader's attention upon the class of problem being illustrated.

1) (INVERT '(cond ((< x 5) 1)
 ((> x 10) 2)
 (t 3)))
 'x
 3)

returns *innocent* implying that no value of x would cause the *cond* to return 3 when in fact any value of x between 5 and 10 will cause the *cond* to return 3. This is a simple example of a large class of problems which INVERT does not handle correctly because its analysis does not probe deeply enough into the structure and meaning of the LISP form being inverted.

2) (INVERT '(cond ((< x 5) 1)
 ((> x 5) 2))
 'x
 2)

returns *innocent* whereas

```
(INVERT '(cond ((< x 5) 1.0)
                (> x 5) 2.0)
                'x
                2))
```

correctly returns (*pair* 5 100). The only difference is the two decimal points. The most efficient and simplest solution to this situation is to ensure that all values which enter the KATE system, either through hardware measurements or through the knowledge base be expressed in floating point. If that cannot be guaranteed, then INVERT can be modified to ensure that all numeric equality tests are performed using "=" instead of equal.

3) (INVERT '(< 1 x 10)
 'x
 t))

returns (*pair* 1 100) when it should return (*pair* 1 10). The documentation for INVERT states that it is not capable of handling an inequality unless the Variable being solved for is at the beginning or end of the list of values so in that sense this error is excusable. On the other hand, INVERT should never return incorrect results without telling the user in some way. Our revisions for INVERT permit the Variable of interest to be any place in such a list of values for "<" or ">" so this problem can be considered corrected.

4) (INVERT '(&nd (< x 10)
 (> x 1))
 'x
 t))

returns (*pair* 1 10) just as it should. The alarming fact is that

```
(INVERT '(&nd (< x 10)  

                    (> x 1))  

        'x  

        nil))
```

also returns (*pair* 1 10) which is blatantly incorrect. This is an example of a large class of Boolean expressions for which the value of the Right Hand Side, in this case nil, is not used in the inversion process. This situation must

be corrected in order to have any degree of faith in the results returned by INVERT when any kind of logical expression is involved.

5) (INVERT '(< 1 5 x)
 'x
 t)

returns (*pair* 1 100) which is, once again, incorrect and misleading.

6) (INVERT '(+ 1 x 3 x)
 'x
 10)

returns 10 or whatever the last argument is. This is again caused by an input which is not in a simplified form, but also another case in which the value returned is incorrect and misleading.

7) (INVERT '(< x 10 5)
 'x
 t)

returns (*pair* 0 10) as the range of values for x which will make the expression t when in fact no value of x can make the expression true.

8) (INVERT '(cond (and (> x 0.0) (< x 1.0)) 1.0)
 (and (> x 1.0) (< x 2.0)) 2.0)
 (and (> x 2.0) (< x 3.0)) 1.0))
 'x
 1.0)

returns (*pair* 0 1) which is not really incorrect but is only a partial solution. This example illustrates the fact that INVERT is presently only able to invert monotonically increasing or decreasing step functions. As we shall discuss in a later section, KATE does not currently have a notation for representing multiple subranges of an interval as would be required for the correct inversion of this expression.

Desirable Capabilities for INVERT

The following comments are meant to point out areas where we feel INVERT needs enhancement. We have limited this wish-list to areas we think are practical and might well be encountered in new knowledge bases very similar to those we have seen. We will not

suggest, for example, that INVERT should be able to invert differential or integral equations because it is apparently not necessary to do so in the control environment KATE is expected to work in.

INVERT should be able to work with other relational operators in addition to `<`, `>` and `=`. There is no obvious reason that INVERT should not handle *less than or equal*, *greater than or equal* and *not equal*. In fact, the whole topic of working with intervals in KATE needs to be clarified. There seems to be no distinction between open and closed intervals in KATE. I assume this is because many measurements are analog in nature and are, in fact, not expected to be exact. That approach may lead to problems in the future as other components are encountered, perhaps digital in nature, in which exact equality or inequality is significant.

A closely related criticism is that the only standard notation for representing intervals in KATE is the "*(*pair* low high)*" notation. There are conceivably many situations in which INVERT really needs to return a collection of subintervals and each one may need to be identified as open, closed, half-open, etc. Developing a notation is not complex but I will leave it to others to do so if the examples from the previous section and these comments are sufficient arguments for its desirability. As a consequence of working with collections of subintervals, the function OVERLAP will need to be rewritten. Some work along these lines can be found in my modified code.

Just as the notation and handling of multiple intervals in INVERT is not fully developed, the use of the special value `*INNOCENT*` also causes some difficulties, at least for this investigator. It may well be that within the operation of KATE the following comments are irrelevant, but we found situations in which the returned value of `*INNOCENT*` could mean that either no value of the Variable of interest could satisfy the conditions for inversion or it could mean that any value of the Variable satisfied the conditions. These situations come up only in logical or relational expressions where the truth or falsity of the expression is determined not by the Variable of interest but by other constants or values in the expression. Thus, in some cases, `No` value of VAR could give the expression the appropriate value, whereas in the other cases `Any` value would do. In both of these cases, INVERT currently returns `*INNOCENT*`. These situations need to be checked thoroughly to assure that they do not cause problems in the operation of KATE.

There are no facilities in INVERT for simplifying expressions. Again, simplification has not been necessary up until now but I think it will become more important as knowledge bases are created by programs instead of by engineers. As an example, if asked to invert an expression such as $(+ 2 X X)$ for X , INVERT will return whatever value is passed in as the Left Hand Side when in fact it should be able to easily simplify the expression and then INVERT it. This is a simple example which would not really appear in a knowledge base but if expressions are assembled by KATE and passed to INVERT, it might be possible to generate similar problems where simplification would be necessary. Unfortunately, simplification of algebraic expressions is time-consuming. At some point a decision will have to be made on this trade-off between generality and speed.

The previous section of examples points out that INVERT currently does not generally handle the inversion of lists very well when the Variable being solved for is not at one end of the list. This is easy to fix and some cases have been covered in our modifications to INVERT. Those examples also pointed to several classes of problems in which the value of the right hand side is ignored in inverting logical expressions. This situation must be fixed for inverting all classes of expressions, whether they be logical, arithmetic or based on (in)equalities.

My final comment is really a question about system design. INVERT assembles the inversion of an expression and returns it to ALT-COM. I cannot understand why INVERT does not go ahead and EVALuate subexpressions as they are assembled rather than returning an assembled expression which ALT-COM must then evaluate. This is a minor point but one which aroused my curiosity when trying to understand how INVERT works.

V. Modifications to KDETECT

In the time available for this work, we made several modifications and enhancements to the operation of INVERT. The Appendix contains code listings with modifications to the version of INVERT we started with indicated. Some of the changes were straightforward, are fully operational, and we recommend that they be incorporated into the

current working version of INVERT. Others are not fully operational but will give an indication of how they might be realized in INVERT. We will describe each of these modifications, evaluate its usefulness and point out potential problems.

A very simple modification which we incorporated into our code was to eliminate the final, unnecessary, recursive call to INVERT when its work is completed. We found that the recursive calls to INVERT often terminate by a test in the main COND statement in INVERT of the form ((eq rhs var) 1hs). Noting that this final call was unnecessary and, on many computer architectures, time-consuming, we tested these conditions and conditionally returned 1hs before executing the recursive call.

The function DO-INEQUALITY was modified to handle expressions in which the VAR being solved for appears anywhere in the list of values being checked. For example, our code can invert an expression such as (< 1 x 10) for x whether the expression should be *t* or *nil*. In order to correctly evaluate such expressions, two primary changes were made to DO-INEQUALITY. The first is that the value of the Left-Hand-Side is used to determine whether to determine values of VAR which make the expression true or to return the complement of those values. In order to do this, we used the convention of expressing multiple intervals in the form "(*pair* low-value₁ high-value₁,...low-value_n high-value_n)". The second change was in the way that the other values in the list, other than VAR itself, were treated. These "constant" values are checked independently of the VAR being solved for and then a value of the VAR itself is determined which will make the expression either true or false, depending on the Left-Hand-Side.

The processing of conjunctive conditions was modified primarily by incorporating the value of the Left-Hand-Side into the inversion. The modified code works the same as previously if the Left-Hand-Side is true. If the Left-Hand-Side evaluates to *nil*, the range(s) of solutions is complemented, again using the notation previously described for representing multiple intervals.

The capability to invert expressions incorporating "=" was added to invert with all the generality described for the operations above. For example, inverting (= 2 x (- 4 2) 2) for x with the Left-Hand-Side equal to *t* returns 2 as a result, whereas inverting the same expression for *nil* returns (*pair* 1 2 2 100), indicating any value other than 2 is a solution.

VI. Future Directions

Anyone familiar with the overall theory and operation of model-based expert systems for process control, and in particular KATE, understands the central role that expression inversion plays. The purpose of this work was to understand how INVERT currently operates in KATE, if possible, to improve INVERT, and to make recommendations as to how future work on INVERT should proceed. We conclude this report with some very general comments on future directions.

It is our conclusion that the choice, made several years ago, to have KATE perform its symbolic inversion of control relationships dynamically at execution time was generally a good decision. It has allowed KATE, the form of its knowledge base, and the interaction between system components to evolve gradually just as any research project must. As KATE moves from the lab into production it becomes critical that INVERT perform its job efficiently, smoothly, but most of all, correctly. We have found and pointed out areas in which correctness is not guaranteed. These problems must be corrected.

The organization of INVERT as a decision-tree based, recursive function is efficient and a natural organization for parsing the kind of S-expressions encountered in control relationships in KATE's knowledge base. We looked into the possibility of organizing INVERT along the lines of a theorem-prover, that is, using Unification as a pattern matcher in order to solve for the Variable of interest. There are at least two reasons that this avenue of research was not pursued further. The first is the existing investment in INVERT. INVERT works correctly and efficiently for simple arithmetic expressions and adding other arithmetic capabilities is not difficult. There is not enough potential gain in switching to a theorem-proving approach at this point to justify the development effort. The second factor is speed. For arithmetic expressions and step functions, which appear to be the primary material for INVERT, the theorem-proving approach involves more overhead than the prescriptive, decision-tree approach currently in use. If INVERT were intended to be a general-purpose symbolic inversion program, there would be potential advantages to a theorem-proving approach, but that is not the situation.

The final question we need to address is that of the efficiency and speed of operation of INVERT. We do not know, in fact, how important this issue is or will become as KATE is used with larger and more complex systems. We have no profiling data to indicate that INVERT is a bottleneck in KATE's operation but we do know that INVERT is heavily used both in diagnostics and control and thus make the following comments under the assumption that any speedup in INVERT is desirable. As far as the current form of INVERT, we were only able to make very minor suggestions for improving speed. In order to gain useful speed, the only approach we view as promising is to pre-compute the inverse(s) of an object at the time the frame for that object is defined and to attach that symbolic inversion to the property list of the object. We do not recommend doing this for all objects, only those whose control expressions have a small number (maybe $<= 3$) of Variables which could possibly be solved for. This approach requires more storage but would eliminate repeatedly constructing the symbolic inverse of an object at execution time.

APPENDIX

```
;;; This is a listing of those components of the file KDETECT.LSP
;;; which have been modified. These modifications are located in
;;; \KATE\KDETECT.LSP
```

```
(defun INVERT (rhs var xlhs &aux found found2 constants
                  (lhs (if (numberp xlhs) (float xlhs) xlhs)))
  (cond
    ((or (eq lhs '*innocent*) (eq rhs '*innocent*)) '*innocent*)
    ((null rhs) (error "in invert with nil") nil)
    ((eq rhs var) lhs)
    ((atom rhs) (error "in invert with an atom") rhs)
    ((sym-nlistp rhs) (error "a is neither atom nor list" rhs))
    ((not (deep-find var rhs))
     (error "in invert not deep found") '*innocent*)
    ((not (setq found (sym-some rhs #'(lambda (sub-rhs)
                                         (deep-find var sub-rhs))))))
     (error "in invert: no sub-term with the variable found") rhs)
    ((and (setq found2 (sym-some (cdr found) #'(lambda (sub-rhs)
                                                 (deep-find var sub-rhs))))
          (equal (car found) (car found2)))
     (invert (car found) var lhs))
    ((eq (car rhs) 'cond)
     (do-cond (cdr rhs) var lhs))
    ((and found2 (eq (car rhs) 'and)))
    ;; Handling of "and" with "found2" modified to use LHS value
    >    (if lhs (overlap (mapcar #'(lambda (term)
                                     (cond
                                       ((deep-find var term)
                                        (invert term var t))
                                       ((eval term) t)
                                       (t (list '*pair* 1. 0.)))))))
    >        (cdr rhs)))
    >    (complement var (overlap ;simplify-pairs
    >                      (mapcar #'(lambda (term)
    >                               (cond
    >                                 ((deep-find var term)
    >                                  (invert term var t))
    >                                 ((eval term) t)
    >                                 (t (list '*pair* 1. 0.)))))))
    >                               (cdr rhs))))))
    >    (found2 (error "% invert can't handle `a: variable `a appears
    >                  in more than one term" rhs var)))
    >    (t (setq constants (remove (car found) (cdr rhs)))))
```

```

>; Handling of recursive call modified
>(setq nextlhs
  (case (car rhs)
    (+ (list '- lhs (join-constants '+ constants)))
    (* (list '/ lhs (join-constants '* constants)))
    (- (cond ((eq (car found) (cadr rhs))
               (list '+ lhs (join-constants '+ constants)))
              (t (list '- (join-constants '- constants) lhs
                        ))))
    (/ (cond ((eq (car found) (cadr rhs))
               (list '* lhs (join-constants '* constants)))
              (t (list '/ (join-constants '/ constants) lhs
                        ))))
    (< (do-inequality lhs rhs found constants var t))
    (> (do-inequality lhs rhs found constants var nil)))
>; Handling of "=" added
> (= (do-equality lhs rhs found constants var ))
  (not (list 'not lhs))
  (and (cond((invert-eval(join-constants 'and constants))
             lhs)
             (t '*innocent*)))
    (or (cond((not(invert-eval(join-constants 'or
                                  constants)))
              lhs)
              (t '*innocent*)))
      (statval lhs)
      (quote lhs)
      (cstatus lhs)
      (a//d-cstatus (list 'inverse-a//d-cstatus lhs)))
      (plus-only (if (>= (invert-eval lhs) 0.)
                     lhs
                     '*innocent*))
      (if (do-if (cadr rhs) (caddr rhs) (nth 4 rhs)
                  (car found) (invert-eval lhs)))
        (max (if (>= (invert-eval lhs)
                      (invert-eval (join-constants 'max constants
                        ))))
          lhs '*innocent*))
        (otherwise (error "~%invert can't handle the ~a
                           operator" (car rhs))))))
    (if (eq (car found) var)
      nextlhs
      (invert (car found) var nextlhs))))))
>
>
>
>
```

```

> ;; Handles processing of expressions beginning with =
> (defun DO-EQUALITY ( lhs rhs found constants var &aux (ev-lhs (eval
> (boolify lhs))) others)
>     (cond ((equal (setq others (eval (cons '= constants))) ev-lhs)
>             (car constants))
>           ((and ev-lhs (null others))
>            '*innocent*)
>             (t (complement var (list '*pair* (eval (car constants))
> (eval (car constants)))))))
>
> ;; Modified to handle VAR anywhere in list
(defun DO-INEQUALITY (lhs rhs sublist constants var lessp
    &aux (ev-lhs (eval (boolify lhs))) 2nd?)
  (setq 2nd? (eq (car sublist) (cadr rhs)))
>  (setq last? (equal sublist (last rhs)))
>  (cond ((and 2nd? ; var appears at beginning of list of arguments
>            (> (length constants) 1))
>            (if (eval (cons (car rhs) constants))
>                (invert (list (car rhs) (car sublist) (car constants))
>                         var lhs)
>                '*innocent*))
>            ((and last? ; var appears at end of list of arguments
>                  (> (length rhs) 3))
>                (if (eval (cons (car rhs) constants))
>                    (invert (list (car rhs) (car (last constants))
>                               (car sublist)) var lhs)
>                    '*innocent*))
>            ((> (length rhs) 3) ; not first or last
>             (setq predecessor (cadr
>                   (member (cadr sublist) (reverse constants):test 'equal)))
>             (if (eval (cons (car rhs) constants))
>                 (if ev-lhs
>                     (overlap (list (invert (list (car rhs) (car sublist)
>                                         (cadr sublist)) var lhs)
>                         (invert
>                           (list (car rhs) predecessor
>                                 (car sublist)) var lhs)))
>                     (complement var (overlap (list (invert (list
>                         (car rhs)
>                         (car sublist)
>                         (cadr sublist))
>                         var t)
>                     (invert (list
>                         (car rhs)
>                         predecessor
>                         (car sublist))
>                         var t))))))
>                 '*innocent*)))

```

```

((or (and ev-lhs 2nd? lessp)
      (and ev-lhs (not 2nd?) (not lessp))
      (and (not ev-lhs) (not 2nd?) lessp)
      (and (not ev-lhs) 2nd? (not lessp)))
      `(*pair* ,(lower-bound var) ,(car constants)))
      (t `(*pair* ,(car constants) ,(upper-bound var)))))

>; Given a VAR, returns complement of the ranges passed in range
>; range can be of form (*pair* low high low high ... )
>(defun COMPLEMENT (var range)
>  (cond ((equal range '*innocent*) '*innocent*)
>        ((and (listp range)
>              (oddp (length range))
>              (eq (car range) '*pair*)
>              (>= (length range) 3 )))
>        (setq range (sort (cdr range) '=< ))
>        (setq result '(*pair*))
>        (if (< (lower-bound var) (car range))
>              (setq result (append result (list (lower-bound var)
>                                               (car range))))))
>        (pop range)
>        (do* ((nextlower (pop range) (pop range))
>              (nextupper (pop range) (pop range)))
>              ((null nextupper)
>               (if (> (upper-bound var) nextlower)
>                   (setq result (append result (list nextlower
>                                         (upper-bound var))))))
>               result))
>        (setq result (append result (list nextlower
>                                         nextupper))))))
>        (t (error "COMPLEMENT cant handle `a' range")
> range)))

>; Turn list of disjoint pairs into a flat list
>(defun SIMPLIFY-PAIRS (pairs)
>  (if (equal pairs '*innocent*)
>      '*innocent*
>      (append '(*pair*)
>              (remove '*pair* (cdr (flatlist pairs)) :test 'equal)))))


```

REFERENCES

- [Calm82] J. Calmet, ed., *Computer Algebra*, Springer-Verlag, New York, 1982.
- [Char83] B. Char, K. Geddes, and G. Gonnet, "Maple User's Manual", University of Waterloo Research Report CS-83-41, December, 1983.
- [Char84] B. Char, K. Geddes, and G. Gonnet, "An Introduction to Maple: Sample Interactive Session", University of Waterloo Research Report CS-84-04, January, 1984.
- [Corn87] M. Cornell, "KATE Software Description", Unpublished Kennedy Space Center Technical Report, January, 1987.
- [Eure87] *Eureka*, Borland International, Scotts Valley, California, 1987.
- [Hulz83] J. vanHulzen, and J. Calmet, "Computer Algebra Systems", in *Computer Algebra: Symbolic and Algebraic Computation*, ed. B. Buchberger, G. Collins, and R. Loos, Springer-Verlag, New York, 1983.
- [MACS83] *MACSYMA Reference Manual*, Version 10, Symbolics, December, 1983.
- [Math87] *MathCAD*, MathSoft Inc., Cambridge, Massachusetts, 1987.
- [Rand84] R. Rand, *Computer Algebra in Applied Mathematics: An introduction to MACSYMA*, Pitman Publishing, Boston, 1984.
- [Scarl87] E. Scarl, J. Jamieson, and C. DeLaune, "Diagnosis and Sensor Validation through Knowledge of Structure and Function", *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-17(3), May/June 1987.
- [Stein88] S. Steinberg, "Overview of Mathematical Symbol Manipulation", in *CWI Quarterly*, 1:3, Amsterdam, September, 1988.
- [Wilf82] H. Wilf, "The Disk With the College Education", *American Mathematical Monthly*, 89:1, January, 1982.
- [Yun80] D. Yun, and D. Stoutmeyer, "Symbolic Mathematical Computation", *Encyclopedia of Computer Science and Technology*, 15, Marcel Dekker, New York, 1980.